COP 5536 Spring 2023
Programming Project Report

Pawan Kumar Jagadapuram

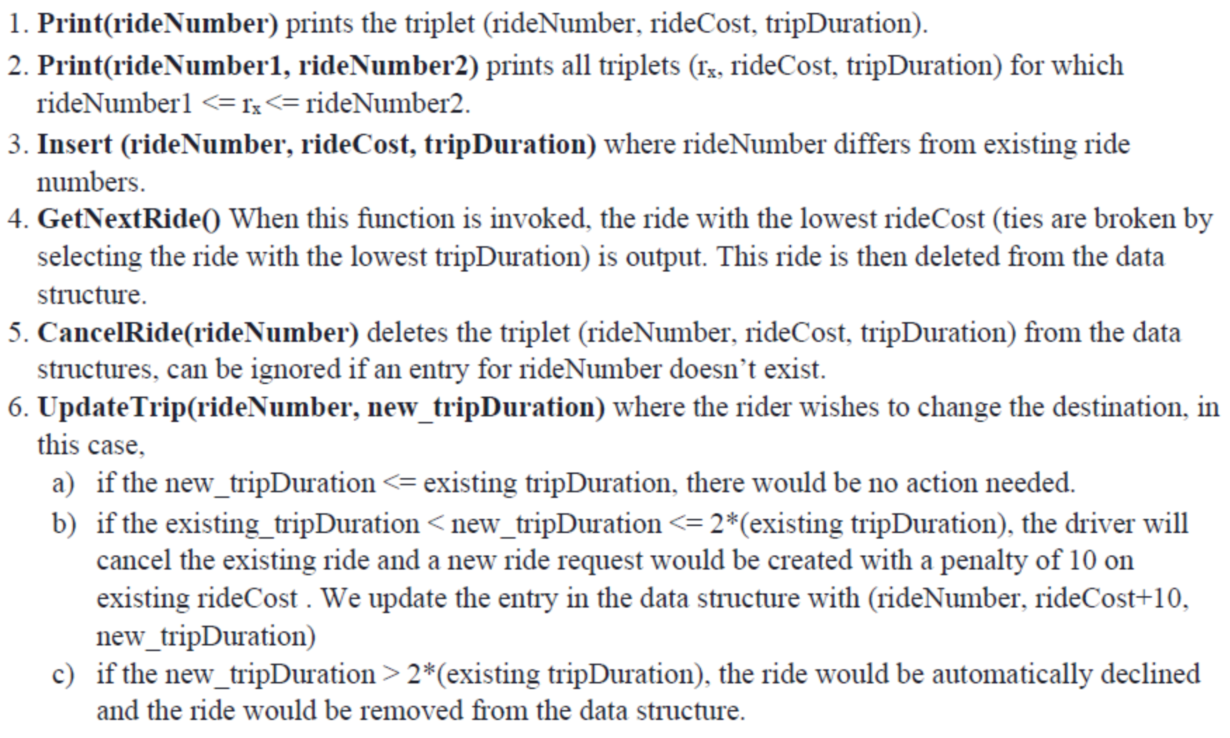
UFID: 73643747

pjagadapuram@ufl.edu

Problem description:

We need to implement a GatorTaxi application where each ride has the following details:
rideNumber, rideCost, tripDuration

Below are the needed operations:



1. **Print(rideNumber)** prints the triplet (rideNumber, rideCost, tripDuration).
2. **Print(rideNumber1, rideNumber2)** prints all triplets (r_x, rideCost, tripDuration) for which rideNumber1 <= r_x <= rideNumber2.
3. **Insert (rideNumber, rideCost, tripDuration)** where rideNumber differs from existing ride numbers.
4. **GetNextRide()** When this function is invoked, the ride with the lowest rideCost (ties are broken by selecting the ride with the lowest tripDuration) is output. This ride is then deleted from the data structure.
5. **CancelRide(rideNumber)** deletes the triplet (rideNumber, rideCost, tripDuration) from the data structures, can be ignored if an entry for rideNumber doesn't exist.
6. **UpdateTrip(rideNumber, new_tripDuration)** where the rider wishes to change the destination, in this case,
   a) if the new_tripDuration <= existing tripDuration, there would be no action needed.
   b) if the existing_tripDuration < new_tripDuration <= 2*(existing tripDuration), the driver will cancel the existing ride and a new ride request would be created with a penalty of 10 on existing rideCost . We update the entry in the data structure with (rideNumber, rideCost+10, new_tripDuration)
   c) if the new_tripDuration > 2*(existing tripDuration), the ride would be automatically declined and the ride would be removed from the data structure.

We are required to implement the above functions using a min-heap and a Red Black Tree data structures.

Min-heap Data Structure: Ordered by rideCost

A min-heap is a binary tree data structure where the value of each node at every level is less than the value of its children. Therefore, the root node always has the minimum value. If implemented in an array the root lies at index position 0. It's left child lies at index position 1 (2*i+1) and its right child lies at index 2 (2*i+2). In this project we have implemented min-heap data structure using array of rides. We defined a ride structure with elements rideNumber, rideCost and rideDuration.

```
struct Ride //Create a Ride data structure
{
    int clr=1;
    int rideNumber, rideCost, rideDuration, heapIndex = -1;
    Ride *parent, *lchild, *rchild;
};
```

We have defined the maximum capacity of the heap to be 5000 rides and also defined few important functions in this min-heap class. Below are the functions:

minHeapify(int)

```
    void minHeapify(int x){   //Heapify function
        int l = lchild(x);
        int r = rchild(x);
        int least = x;
        if (l < hp_size && comp(heap_arr[x],heap_arr[l]))
            least = l;
        if (r < hp_size &&  comp(heap_arr[least],heap_arr[r]))
            least = r;
        if (least != x)
        {
            heap_arr[x]->heapIndex = least;
            heap_arr[least]->heapIndex = x;
            RidePtr tmp= heap_arr[x];
            heap_arr[x] = heap_arr[least];
            heap_arr[least] = tmp;
            minHeapify(least);
        }
    }
```

The above function performs minheapify operation on the min-heap. It is used to maintain the min-heap property in the data structure by updating the node with the small element (ride with least cost) to the root of the subtree, recursively applying the same operation until the min heap property is satisfied i.e. every parent node is less than it’s right and left children.

extractMin()

```
RidePtr extractMin(){      // extract min element in the min heap
    if (hp_size <= 0)
        return NULL;
    if (hp_size == 1)
    {
        hp_size--;
        return heap_arr[0];
    }
    RidePtr root = heap_arr[0];
    heap_arr[0] = heap_arr[hp_size-1];
    heap_arr[0]->heapIndex = 0;
    hp_size--;
    minHeapify(0);

    return root;
}
```

The above function is used to extract the ride node with the least rideCost. This function is called by the getNextRide() function, which will extract the ride with the least rideCost currently in the tree. Once the next ride is removed from the data structure, the root node is first updated with the last element in the data structure and then minHeapify() function is invoked to maintain the min-heap property such that every parent node in the tree is less than it’s children.

insertKey(Rideptr)

```
void insertKey(RidePtr m){  //Insert elements in min heap
    if (hp_size == max_size)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }
    hp_size++;
    int x = hp_size - 1;
    heap_arr[x] = m;
    m->heapIndex = x;
    while (x != 0 && comp(heap_arr[parent(x)],heap_arr[x]))
    {
        heap_arr[x]->heapIndex = parent(x);
        heap_arr[parent(x)]->heapIndex = x;
        RidePtr tmp= heap_arr[x];
        heap_arr[x] = heap_arr[parent(x)];
        heap_arr[parent(x)] = tmp;
    x = parent(x);
    }
}
```

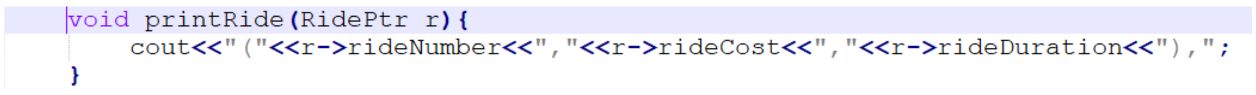
This function is called by both the Insert() and UpdateRide() functions. This function is used to insert the elements in the min-heap by first placing the element at the end of the heaparray. Then a while loop is used to check for the inserted element’s parent value and compare it. If the parent value is greater than the child value then we swap the elements such that we maintain the min-heap property. This while loop runs until all the parent node values are less than its children. Ultimately, the root node has the least rideCost value.

Red Black Tree Data Structure: Ordered by rideNumber

A Red Black Tree is a self-balancing Binary Search Tree (left subtree is less than the parent node and right subtree should be greater than the parent node) with each node colored either red or black. By implementing a RBT we perform all operations like insertion, deletion, searching which would take O(log(n)) time.

Each node in the RBT is represented using a RidePtr structure with attributes rideNumber, rideCost, rideDuration and color (1 for red and 0 for black). We have defined the following few important functions in the RBT class.

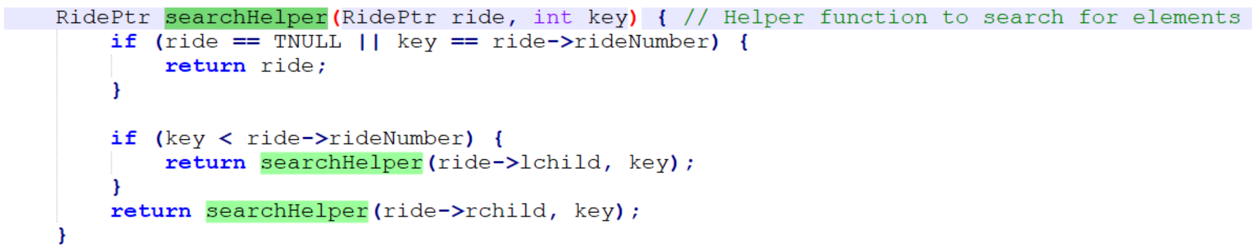
printRide(RidePtr)



```
void printRide(RidePtr r){
    cout<<"("<<r->rideNumber<<","<<r->rideCost<<","<<r->rideDuration<<"),";
}
```

This function prints the details of the given ride node.

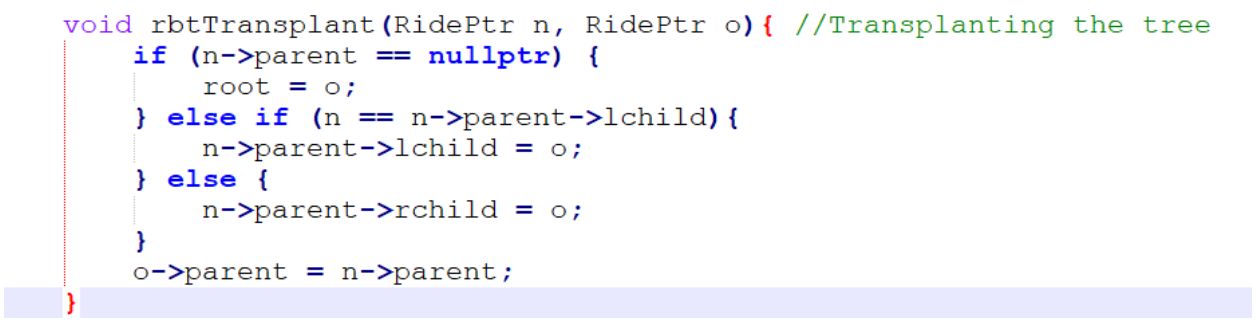
searchHelper(RidePtr, int)



```
RidePtr searchHelper(RidePtr ride, int key) { // Helper function to search for elements
    if (ride == TNULL || key == ride->rideNumber) {
        return ride;
    }

    if (key < ride->rideNumber) {
        return searchHelper(ride->lchild, key);
    }
    return searchHelper(ride->rchild, key);
}
```

This helper function is used recursively search for a node with a given keyValue (rideNumber) in the tree. Since we have implemented the RBTree ordered by the rideNumber, this will only search either the left subtree or the right subtree making the time-complexity for searching as O(log(n)).

rbtTransplant(RidePtr, RidePtr)



```
void rbtTransplant(RidePtr n, RidePtr o){ //Transplanting the tree
    if (n->parent == nullptr) {
        root = o;
    } else if (n == n->parent->lchild){
        n->parent->lchild = o;
    } else {
        n->parent->rchild = o;
    }
    o->parent = n->parent;
}
```

The Transplant function is invoked after every deletion of a node in the RBT to ensure the RBT properties are not violated.

We have also implemented the below functions in the RBT:

updateInsert(RidePtr m): A function that fixes the tree after a node is inserted into the tree. This function also ensures that the properties of the Red-Black Tree are maintained whenever insertion happens.

rideDeleteHelper(RidePtr ride, int key): A helper function that deletes the node with the given key value (rideNumber) from the tree. The function searches for the node using searchHelper() and then uses fixingDelete() to fix the tree after deletion.

Detailed analysis of some of the key functions that are called in the main function are below:

Key functions:

Insert(rideNumber, rideCost, rideDuration)

Print(rideNumber)

Print(rideNumber1, rideNumber2)

UpdateTrip(rideNumber, new_TripDuration)

CancelRide(rideNumber)

GetNextRide()

Insert(rideNumber, rideCost, rideDuration, Minheap, RBTree):

```
void Insert(int rideNumber, int rideCost, int rideDuration,  MinHeap *h, RBTree *rbt){
    RidePtr temp = rbt->search(rideNumber);
    if(temp!=NULL && temp->rideNumber == rideNumber){
        cout<<"Duplicate RideNumber";
        exit(1);
    }
    RidePtr r = createRide(rideNumber,rideCost,rideDuration);
    h->insertKey(r);
    rbt->insert(r);
}
```

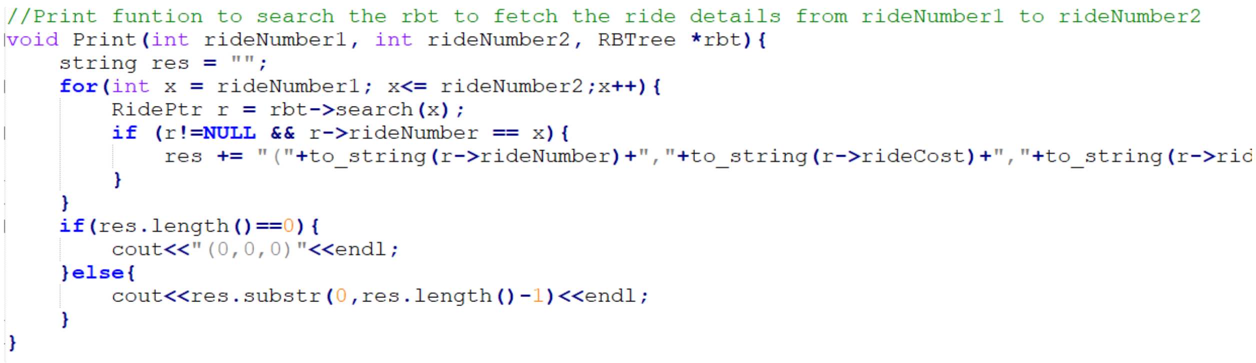
The above function is called whenever the program encounters Insert() command in the input file file_name.txt The above function checks if there already exists a ride with the rideNumber, if yes, then it displays Duplicate RideNumber in the output and terminates the program. If no, then the function calls insertKey() and insert() functions of the min-heap and rbt to insert the new ride in the respective data-structures. Working of insertKey is defined above in the Min-Heap section and it takes O(log(n)) times to insert the key at its position. Because in the worst case the ride value must be checked with nodes at most the height of the min-heap which could be n. Therefore, the time complexity of this function on O(log(n))

Print(rideNumber, RBTree):

```
void Print(int rideNumber, RBTree *rbt){
    RidePtr r = rbt->search(rideNumber);
    if (r!=NULL && r->rideNumber == rideNumber){
        printRide(r);
        cout<<endl;
    }
    else{
    cout<<"(0,0,0)"<<endl;
    }
}
```

This function is called whenever the program encounters Print(rideNumber) command in the input file file_name.txt. The above function inturn invokes the search() function which later calls the searchHelper() function of the RBT class. The searchHelper function details are specified in the RBT section above. While searching for a ride in the Red Black Tree we only travel either the left subtree or the right subtree depending on the value of the rideNumber. In the worst case the rideNumber could be the leaf node in the data structure which makes the time-complexity of this function to be O(log(n)).

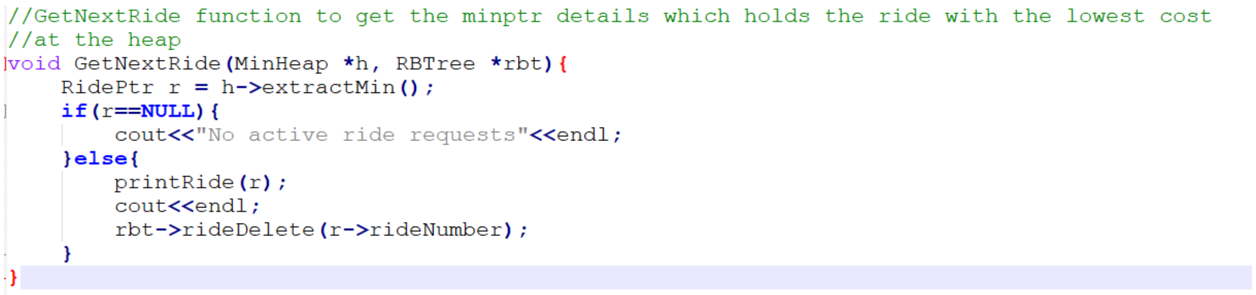
Print(rideNumber1, rideNumber2, RBTree):



```
//Print funtion to search the rbt to fetch the ride details from rideNumber1 to rideNumber2
void Print(int rideNumber1, int rideNumber2, RBTree *rbt){
    string res = "";
    for(int x = rideNumber1; x<= rideNumber2;x++){
        RidePtr r = rbt->search(x);
        if (r!=NULL && r->rideNumber == x){
            res += "("+to_string(r->rideNumber)+","+to_string(r->rideCost)+","+to_string(r->ric
        }
    }
    if(res.length()==0){
        cout<<"(0,0,0)"<<endl;
    }else{
        cout<<res.substr(0,res.length()-1)<<endl;
    }
}
```

The above Print function is called when program encounters Print(rideNumber1, rideNumber2) command in the input file file_name.txt. This function also searches the RBT and outputs the rides that are available between the rideNumber1 and rideNumber2 (both inclusive). If there are no rides specified in the range then it outputs (0,0,0). This function searches the tree at O(log(n+S)) time in the worst case where S is the number of rides outputted.

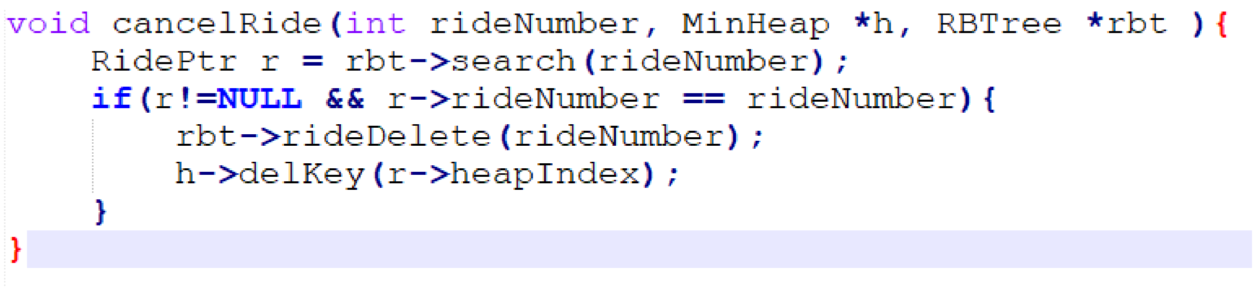
GetNextRide(MinHeap, RBTree):



```
//GetNextRide function to get the minptr details which holds the ride with the lowest cost
//at the heap
void GetNextRide(MinHeap *h, RBTree *rbt){
    RidePtr r = h->extractMin();
    if(r==NULL){
        cout<<"No active ride requests"<<endl;
    }else{
        printRide(r);
        cout<<endl;
        rbt->rideDelete(r->rideNumber);
    }
}
```

The above function is called when program encounters getNextRide() command in the input file file_name.txt. The above function is supplied with the root nodes of the min-heap and the RBT. Since the min-heap is ordered using the rideCost, getNextRide() takes O(1) to get the details of the next available ride. However, after deleting the ride with the least rideCost, we need to ensure the min-heap and RBTree properties are satisfied. Therefore, we perform Heapify and rotation operations in the min-heap and RBTree which takes O(log(n)) in the worst-case scenario. Therefore, the overall time complexity of this function is O(log(n)).

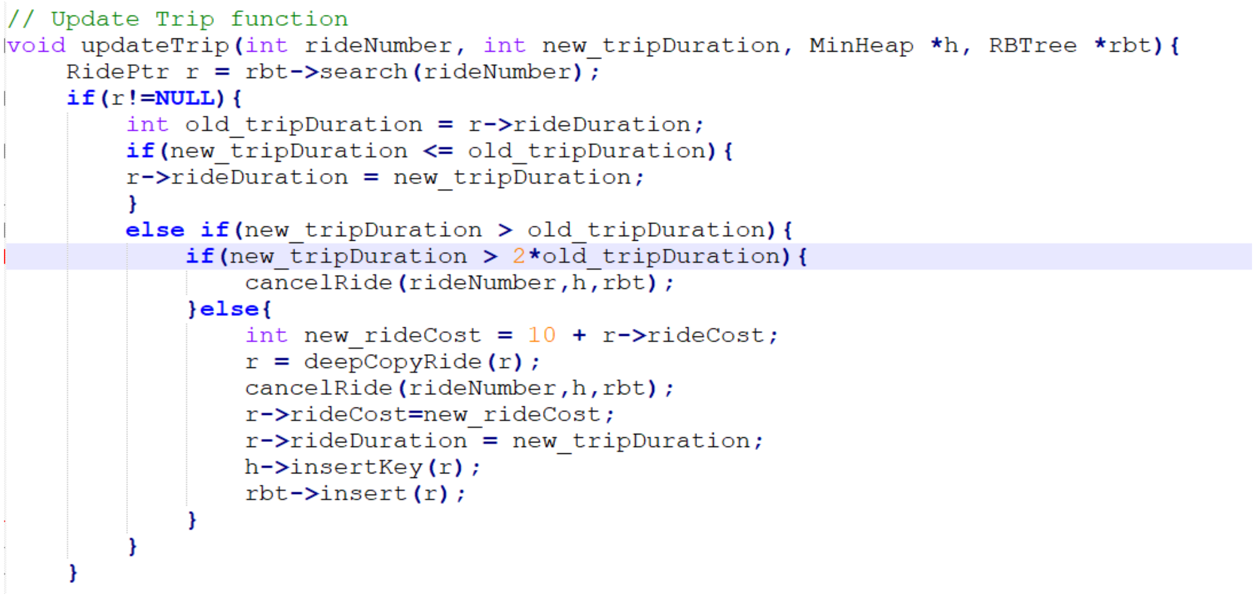
cancelRide(rideNumber, MinHeap, RBTree):



```
void cancelRide(int rideNumber, MinHeap *h, RBTree *rbt ){
    RidePtr r = rbt->search(rideNumber);
    if(r!=NULL && r->rideNumber == rideNumber){
        rbt->rideDelete(rideNumber);
        h->delKey(r->heapIndex);
    }
}
```

The above function is called whenever program encounters CancelRide(rideNumber) command in the input file file_name.txt. The above function takes rideNumber, min heap pointer and RBTree pointer as the input parameters and calls the search function of the RBTree class. The search function takes O(log(n)) time to search for the ride specified in the worst case and if the ride is found then this function invokes rideDelete() and delKey functions() of the RBTree and minHeap classes to perform deletion operation of the ride. Deletion operation also take O(log(n)) time in the min heap and the RBTree and therefore the overall time-complexity is O(log(n)) for this function.

updateTrip(rideNumber, new_TripDuration, MinHeap, RBTree):



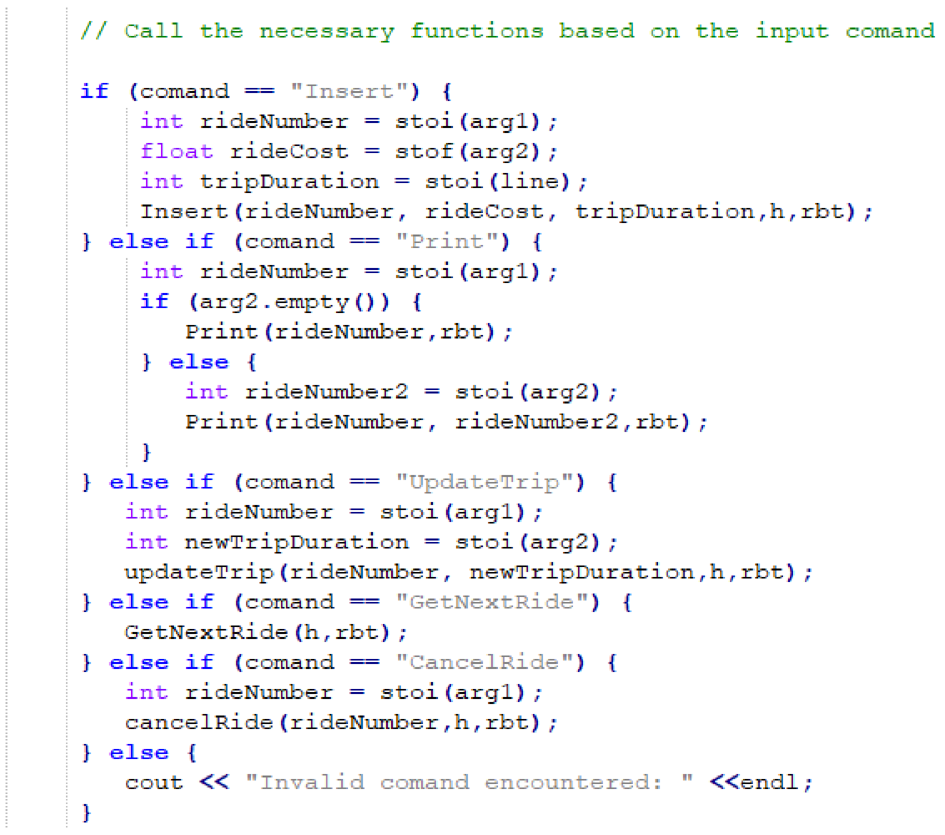
```
// Update Trip function
void updateTrip(int rideNumber, int new_tripDuration, MinHeap *h, RBTree *rbt){
    RidePtr r = rbt->search(rideNumber);
    if(r!=NULL){
        int old_tripDuration = r->rideDuration;
        if(new_tripDuration <= old_tripDuration){
        r->rideDuration = new_tripDuration;
        }
        else if(new_tripDuration > old_tripDuration){
            if(new_tripDuration > 2*old_tripDuration){
                cancelRide(rideNumber,h,rbt);
            }else{
                int new_rideCost = 10 + r->rideCost;
                r = deepCopyRide(r);
                cancelRide(rideNumber,h,rbt);
                r->rideCost=new_rideCost;
                r->rideDuration = new_tripDuration;
                h->insertKey(r);
                rbt->insert(r);
            }
        }
    }
```

The above function is invoked whenever program encounters UpdateTrip(rideNumber, new_TripDuration) command in the input file

file_name.txt. The above function will check if the new_TripDuration is less than the old_TripDuration of the given rideNumber. If yes, then it will update the rideDuration with the new_TripDuration of the ride at the given rideNumber. If not, then it will check for the condition if the new_TripDuration is > 2*(existing duration). If yes, then it will cancel the ride. If no, then it will update the ride with the new_TripDuration and also update the rideCost with a summation of +10. The above functions invoke the search() and cancelRide() functions to do the necessary operations which both take O(log(n)) time. Therefore, the overall time complexity for this function is O(log(n)).

Main function:

Main function will read the command line arguments which is supplied with the input file with the commands. The function opens the input.txt file and reads each line in a command and call the necessary functions.



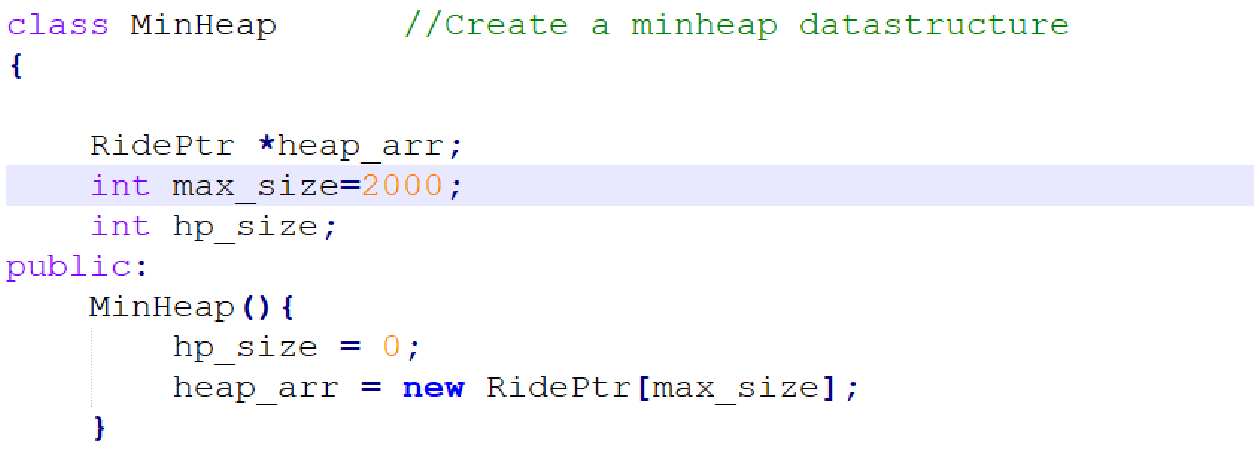
```
// Call the necessary functions based on the input comand

if (comand == "Insert") {
    int rideNumber = stoi(arg1);
    float rideCost = stof(arg2);
    int tripDuration = stoi(line);
    Insert(rideNumber, rideCost, tripDuration,h,rbt);
} else if (comand == "Print") {
    int rideNumber = stoi(arg1);
    if (arg2.empty()) {
       Print(rideNumber,rbt);
    } else {
       int rideNumber2 = stoi(arg2);
       Print(rideNumber, rideNumber2,rbt);
    }
} else if (comand == "UpdateTrip") {
   int rideNumber = stoi(arg1);
   int newTripDuration = stoi(arg2);
   updateTrip(rideNumber, newTripDuration,h,rbt);
} else if (comand == "GetNextRide") {
   GetNextRide(h,rbt);
} else if (comand == "CancelRide") {
   int rideNumber = stoi(arg1);
   cancelRide(rideNumber,h,rbt);
} else {
   cout << "Invalid comand encountered: " <<endl;
}
```

Also, the result of the code is returned in the file name called output_file.txt.

In the end both the input and output files are closed by the program.

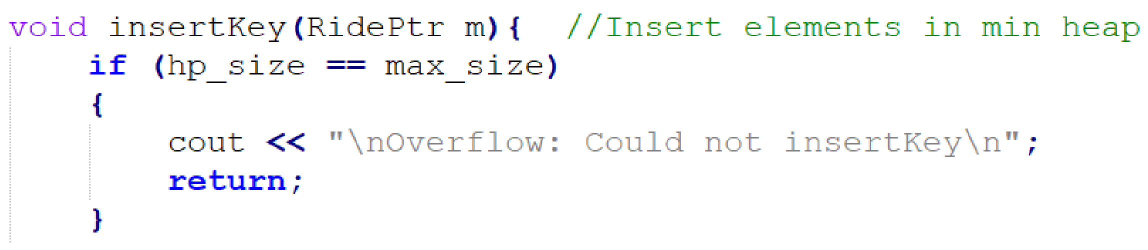
Space Complexities of minHeap and RBTree:

In our program we have defined a minHeap as an array of ride structures and the max size of the array we have given is 2000. That means the heap can store at maximum 2000 rides at its fullest capacity. However, the heap data structure is dynamically allocated using the new operator as and when a new ride is created. Therefore, the space complexity is O(n) where n is the input rides.



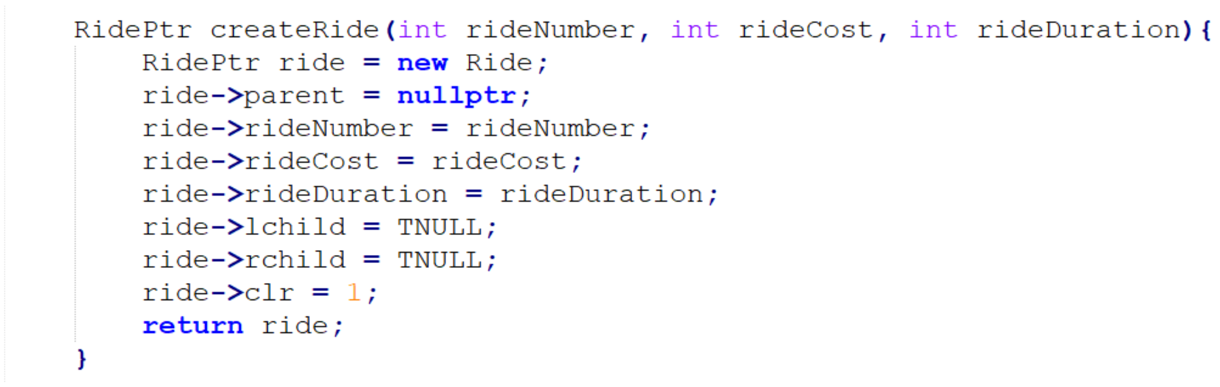
```
class MinHeap      //Create a minheap datastructure
{

    RidePtr *heap_arr;
    int max_size=2000;
    int hp_size;
public:
    MinHeap(){
        hp_size = 0;
        heap_arr = new RidePtr[max_size];
    }
```

If when trying to insert an element after the heap reached its maximum capacity, the program will output that the heap is overflowing.



```
 void insertKey(RidePtr m){  //Insert elements in min heap
     if (hp_size == max_size)
     {
         cout << "\nOverflow: Could not insertKey\n";
         return;
     }
```

The space complexity of the RBTree is O(n) where n is the number of active rides. Each ride takes a constant space for the rideNumber, rideCost, rideDuration, parent ptr, lchild ptr, rchild ptrand color flag which is O(1) Therefore, the space complexity of the RBTree is O(n) where n is the number of active rides.



```
RidePtr createRide(int rideNumber, int rideCost, int rideDuration){
    RidePtr ride = new Ride;
    ride->parent = nullptr;
    ride->rideNumber = rideNumber;
    ride->rideCost = rideCost;
    ride->rideDuration = rideDuration;
    ride->lchild = TNULL;
    ride->rchild = TNULL;
    ride->clr = 1;
    return ride;
}
```

Learning outcome: By doing this project, we have learned the implementation of advance data structures like min heaps and RBTrees which are highly efficient whenever there is frequent insertion, deletion and searching operations that are needed to be perform.